UCRL-MA-103421 Rev. 3

Security Profile Inspector for the UNIX Operating System (SPI/VMS)

User's Guide

for SPI/VMS version 2.0

Tony Bartoletti
Steve Cooper
John Fisher
Susan Taylor

February 1995

Lawrence Livermore National Laboratory

SPI is sponsored by:
US Air Force Cryptologic Support Center
US Department of Energy
US Defense Information Systems Agency

Work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

ACKNOWLEDGMENT:

The work reported in the following pages has been sponsored by several organizations. We appreciate the support, advice, and technical direction of:

U. S. Department of Energy
Office of Safeguards and Security
SA-123/GTN
Washington, DC

U. S. Department of Defense
Defense Information Systems Agency
ATTN: TFE
Arlington, VA

U. S. Air Force
Cryptologic Support Center
AFCSC/SRE
San Antonio, TX

Table of Contents

Introduction

Welcome to SPI/VMS 2.0. SPI is a software security package for computers running the UNIX or VMS operating systems. Its purpose is to assist system managers and computer security officers in insuring the security of their computer systems. SPI inspects various aspects of the computer system and reports on potential vulnerabilities.

Computer system vulnerabilities are weaknesses in a system's defenses against intruders. Examples of vulnerabilities include: improper world permissions, easy-to-guess passwords, or accounts that no longer have users. SPI helps to detect these vulnerabilities so that you can eliminate them.

Computer intrusions are detrimental changes to a system. SPI helps to detect these changes by taking an initial "snapshot" of the system configuration, and then using this as a baseline for comparison when you need to check what elements have been affected by an intrusion. Intrusions are such things as: a changed digital signature on a file that should have been left alone, unexpectedly changed file permissions or ownerships, new or modified user accounts, and corrupted log data.

SPI VMS contains four powerful security tools:

1) Quick System Profile - for exploring known security problems, such as incorrect permissions on important files, or dangerous settings in configuration files.

2) Password Security Inspector - uncovers poorly choosen passwords, using a dictionary and common permutations of user information.

3) Change Detector - maintains a database of important system file statistics, such as file size and the last date changed, and provides warnings when that information has been changed.

4) Configuration Query Language - a scripting language that provides a high level method of analyzing the target system. Each of the three tools above utilize CQL to capture system information, and the user is encouraged to write more scripts that check for site-independent problems.

In addition, SPI VMS provides the following for managing those tools:

1) Menu-based interface - provides an easy-to-use interface for running SPI tools.

2) Report Generator - provides user-friendly reports, generated from the data files produced by the security tools.

SPI/VMS 2.0 is a complete rewrite of SPI/VMS 1.0. This new verions is directly derived from SPI/UNIX 3.2.1, and shares most of the same code.

Installation

Setting up SPI VMS 2.0 requires several steps:

1) Execute the following command to build the SPI hierarchy:

```
BACKUP SPI.BCK/SAVE [SPI...]
```

where [SPI] is the directory you wish SPI to reside in. The following directories will now be created:

`[SPI.E]`	All executables
`[SPI.D.PARAMETERS.CDT]`	Location of `METASPEC.CDT`, for customizing the Change Detector
`[SPI.D.PARAMETERS.RG]`	Report formats for the report generators
`[SPI.D.DATABASE.PSI]`	Location of dictionaries for the Password Inspector
`[SPI.D.DATABASE.CDT]`	Database information stored by the Change Detector and the Password Inspector
`[SPI.D.CORF]`	Raw output generated by the security tools.
`[SPI.D.RESULTS]`	User-friendly output reports, generated from the CORF output.

3) Edit the SPI DCL file `[SPI.E]SPI.COM`, setting the logical `SPIDIR` so that all directories point to `[SPI]` (or wherever you choose SPI to reside).

Using SPI

The SPI interface itself is quite straightforward. To start it, type the following:

```
@SPI
```

The following menu will then be presented:

```
---------------------
SPI/VMS 2.0 MAIN MENU
---------------------

1   Quick System Profile
2   Password Security Inspector
3   Change Detector Snapshot
4   Change Detector
5   CQL Script
X   Exit

Option:
```

When a tool is executed, it will run in the background, and the user may continue, even leaving SPI. When the tool has completed, a textual message will be sent to the screen. Multiple tools may be run concurrently, but only one copy of each tool is allowed.

To execute a tool in the background, type

```
SPI nn
```

where `nn` is the menu item to execute (1-4 are valid). This will execute the tool without running the SPI interface.

After the intial execution of SPI, the following logicals will be created:

`SPI$QSPRESULT` QSP Reports

`SPI$PSIRESULT` PSI Reports

`SPI$CDTRESULT` CDT Reports

`SPI$CQLCORF` CQL CORF output

Each of the tools presented above are detailed in the following sections.

Quick System Profile (QSP)

QSP checks the target system for security problems that are specific to VMS. QSP is actually a CQL script, called `QSCRIPT`.

Currently, QSP is limited to checking file permissions on key configuration files, but QSP will later be expanded to deal with more VMS-specific security problems.

Starting QSP

To run QSP, execute the SPI DCL, and choose option 1.

When QSP has finished, the report may be found in `SPI$QSPRESULT`, under the name `QSP`{time}, where {time} is the date and time that the report was generated.

Command Line Options

QSP contains no command line options.

QSP Analysis

QSP utilizes CQL (Configuration Query Language) to investigate known VMS security problems. These problems are outlined below.

File Permissions

The following files should be readable only by system administrators:

```
SYS$SYSTEM:AUTHORIZE.EXE
SYS$SYSTEM:NETUAF.DAT
SYS$SYSTEM:NETPROXY.DAT
SYS$SYSTEM:SYSUAF.DAT
SYS$SYSTEM:FTSVQUEUE.DAT
SYS$SYSTEM:FTSVACC.DAT
SYS$MANAGER:VAXNOTES$STARTUP.COM
SYS$MANAGER:LOGIN.COM
```

The following files should be executable only by system administrators:

```
SYS$SYSTEM:TDMSEDIT.COM
SYS$SYSTEM:TMDSTRTUP.COM
SYS$SYSTEM:MODPARAMS.DAT
SYS$MANAGER:SYSSHUTDWN.COM
SYS$MANAGER:SYLOGIN.COM
SYS$MANAGER:SYSTARTUP.COM
SYS$MANAGER:SYSTARTUP_V5.COM
SYS$MANAGER:LOGIN.COM
```

```
SYS$MANAGER:STARTNET.COM
SYS$MANAGER:LOADNET.COM
SYS$MANAGER:RTTLOAD.COM
```

Password Security Inspector (PSI)

The Password Security Inspector checks the password file for easily guessed passwords.The password file contains information such as account names, passwords, project names, etc, that usually contains information about the user. SPI checks passwords in the password file against this invormation and various dictionaries, using certain algorithmic permutations, to find passwords that are easy-to-guess or find.

PSI is the password security inspector; it will perform a series of tests on /etc/passwd or any other file written in the same format. psi will attempt to match the passwords in the tested file against one to three specified dictionaries and your account name plus, all possible combinations of one, two, and three letter alpha and numeric passwords. psi can, during the tests, reverse dictionary words and capitalize them. (accountname and gecos field entries are automatically reversed and uppercased.)

Running PSI

To run PSI, execute the SPI.COM DCL, and choose option 2.

When PSI has finished, the report may be found in `SPI$PSIRESULT`, under the name PSI{time}, where {time} is the date and time that the report was generated.

Without modification, PSI runs with no special options (e.g, no dictionaries). To add options, edit the `SPAWNSPI.COM` DCL, where PSI is executed, using the following options:

Command Line Options

```
psi   [-h][-f][-r] [-d dictionary(s)]
      [-a alpha_num][-l logfile][-o agelimit]
```

`-h`	display command line usage
`-f`	Print the bad passwords when found.
`-r`	Reverse the dictionary test words (e.g. "secret" becomes "terces").
`-d dictionary(s)`	List of dictionary files, each file consisting of one testword per line.
`-a alpha_num`	Extent of alpha and numeric test words to generate. "alpha_num" must be either 1, 2, or 3.
`-l logfile`	Logfile to use and update. When a logfile is specified, only those accounts whose passwords have changed from the logfile record will be tested, and the logfile will be updated to reflect the current entries. Also, new and deleted accounts will be reported.

	If no logfile is given, then all account names will be tested, and no subsequent logfile will be produced.
`-o agelimit`	Report accounts whose password has not changed in "agelimit" days. This option requires the "-l logfile" option. psi outputs its results to standard output. You may also configure and run psi through the spi menu package.

PSI Analysis

PSI checks for the following types of passwords:

- empty passwords
- 1-, 2-, or 3-letter combinations of alphabetic and numeric permutations
- variations of the user’s name or information fields.
- common English words, local jargon, and trivial words (e.g., names of characters in books or names of cars)

Change Detector (CDT)

CDT is used to track changes to important system files, user accounts, and related system security attributes. The purpose is both to detect intrusions and to warn when the attributes of system-critical files may have been inadvertently and detrimentally modified during routine administrative actions.

The SPI Change Detector Tool (CDT) is used to track suspicious changes to important system files, user/group accounts, and related system security attributes. It is designed to allow flexible specification of those items warranting change detection. The cdt utility is used both to conduct change detection, and to update the baseline (cdt database) of previously accepted system values with new values (as systems are upgraded, users and critical files are added or deleted, etc).

Starting CDT

CDT works by saving a "snapshot" of the system in a database, and alerting the system administrator when the the system has changed.

The criteria for the snapshot lies in `SPI$CDTPARAMS.` See that document for information on how to modify the system snapshot.

To create a system snapshot, execute the `SPI.COM` DCL, and choose option 3. No report is produced, but a database will be created called `SPI$CDTDATA:DATABASE.`.

To see how the system has changed since the snapshot, execute the `SPI.COM` DCL, and choose option 4.

When CDT has finished, the report may be found in `SPI$CDTRESULT`, under the name `PSI`{time}, where {time} is the date and time that the report was generated.

Command Line Options

```
cdt [-h] | [-u] -d database_ptr [-p paramfile | -s snapshot]
```

`-d database`	file representing the active cdt database (mandatory - see FILES below)
`-h`	display command line usage
`-p paramfile`	file specifying the items for which change detection is to be reported, or (if -u option given) the items which will be gathered to populate the baseline cdt database. The paramfile is used by cdt to automatically generate a snapshot of current system values. If [-p paramfile] is omitted, then [-s snapshot] is required. (see FILES below for detailed paramfile format)
`-s snapshot`	file containing current values of user, group and file lists and their attributes. The values contained will be compared against corresponding data in the cdt database so that changes can be

reported, or (if -u option given) these new values will be used to populate (update) the cdt database. If [-s snapshot] is omitted, then [-p paramfile] is required to produce the needed snapshot. (see FILES below)

`-u` Update Database (no change detection) -- Create revised CDT database. All records in the database file whose sixth field is "CDT" are deleted, and replaced by new entries. The new entries are generated either from a supplied "snapshot" file, or from a snapshot produced by a call to the SPI Query Language utility (sql) parametrized by a supplied "paramfile". No change detection is conducted.

With any update of a CDT database, the previous version is saved in the same directory with a name of the form "sdb_YYYYMMDD.HHMMSS", where the date.timestamp will be the latest creation or modification time.

NOTE: A change-detection report will only be issued if the -u option is omitted.

CDT Analysis

A variety of user and file parameters may be stored in CDT's database. These parameters are stored in METASPEC.CDT. This file is used to specify which files, users, and groups are to be part of the change detection baseline, and for each, which attributes are to be considered significant for change detection reporting. All lines (other than #comment lines) have the form:

<EntryType>:<WarnSpec>:<MetaSpec>[:<ExceptionSpec>]

Example -- to do change detection of all files in /etc for changes in group_id, linkcount, permissions, size, user_id, and xsum:

```
FILE:glpsux:/etc/*
```

In the above example, to exclude the file /etc/wtmp,

```
FILE:glpsux:/etc/*:/etc/wtmp
```

Valid Entry_Types are: SYSTEM, HOST, FILE, USER, GROUP

Valid FILE WarnSpecs are: abcdgilmpstux

(a)access_time	(c)change_time	(d)majordevice	(g)group_id
(i)inodenumber	(l)linkcount	(m)modify_time	(p)permissions
(s)size	(u)user_id	(t)xsum_length	(x)xsum

NOTE: to detect changes in file CONTENTS, (x)xsum must be included. The following special FILE WarnSpecs are allowed:

use (A) for acdgilmpstux (Report ALL Changes)

use (L) for gilpu (useful for logfiles and devices)

use (S) for cglpsux (useful STANDARD for most other files)

use (X) for x (XSUM -- crypto checksum only)

You may also use combinations like S+mt and A-bdx, A-L, etc.

Valid USER Warnspecs are:

(p)password (u)user_id (g)group_id (i)info_field
(l)login_dir (s)commandshell

Valid GROUP WarnSpecs are:

(g)group_id (m)memberlist

Configuration Query Language (CQL)

The Configuration Query Language (CQL) is used to detect bad system configuration practices (such as bad permissions for files, or the presence of bad service configurations such as for anonymous ftp). CQL is a language by which one can specify which bad configuration practices one wants to check for.

A very useful set of generic practices to test for is available by running the QSP (Quick System Profile) tool. To create a series of checks specific to the needs of your system, a custom CQL script may be written.

The SPI Configuration Query Language is designed to allow users an easy way to specify and test for configuration practices or policies specific to their systems. As mentioned, the QSP tool is essentialy a CQL script. This default set of tests is very good, however if you would like to tailor this set to your system or write your own, feel free to do so.

Essentially, the CQL language allows you to run a series of configuration tests over sets of files, users, or groups. The tests can include checks for standard information (a file's permission or owner, a user's group, a group's members, etc), and predefined functions. DCL scripts may be incorporated as needed, and C functions may be incorporated into the language for more sophisticated tasks.

Command Line Options

```
cql   [-h] -i filelist [-your_var [value]]
```

`-h`	display command line usage
`-i filelist`	execute the scripts listed in filelist

All - options (except -i, -h) to cql will be treated as the name of an execute line variable. This variable has the value "on", unless a value follows the option. In the query script, you can test if an execute line variable was not present by testing if its value is "off".

Queries

The basic construct of a query is as follows: for a given list of files, users, or groups, perform a set of security violation tests.

Any item (file, user, group) which passed any of the security tests is then reported. The reported information can be either the list of all passing items (keyword WHICH), or a separate report for each item and why it failed (keyword STATUS).

The following examples illustrate this. The first example tests if either of the files `[FRANK]LOGIN.COM` and `[TED]LOGIN.COM` are world writable. If the test passes, then the current permissions is reported. The second example reports which users have bad passwords.

```
STATUS FILES={‘[FRANK]LOGIN.COM ‘[TED]LOGIN.COM}
{
     PERM = w_w
}

WHICH USERS=ALL
{
     EVAL(chk_passwd())=bad
}
```

Query Tests

The set of tests executed for a query is one or more simple, compound, or recursive tests. A simple test is of the form

quantity = value

quantity != value

where the quantity is one of the following:

system variable (i.e. `osname=VMS`)

object variable (i.e. file permission, group number)

execute line variable

`EVAL(“user program”)` (more fully explained later)

`EVAL(user_subroutine())`

and value may be a list. If more than one value is given, then the list must be enclosed by {}’s. For example:

```
OWNER={bad1 bad2}

GROUP!={wheel other}
```

Compound tests can include any combination of the above quantities. For example:

```
system=vms & EVAL(“vms_security_script”)!=0
```

The above would execute a DCL script called vms_security_script and test its exit code against 0. Although, a better way to do the above would be to use an IF command.

```
IF (system=VMS) EVAL(“vms_security_script”)!=0
```

System Variables

System variables allow the user to tailor his checks for one or more systems, thus allowing him to use the same query script for multiple operating systems or architectures.

The following system variable are available:

osname	operating system name (VMS)
osnum	operating system number (6.1)
mach	machine architecture (VAX 8650)
hostname	name of host machine

Object Variables

Object variables are pieces of information about a file, user, or group such as the file owner, or the users belonging to a group. You can use this information to determine if an object is not configured as it should be. For example, a file which should be owned by the system administrator, but is not.

The following variables are available. Examples of their use will be given later.

File Variables

access	time file was late created (format yymmdd.hhmm e.g. 940223.1416)
group	name of the group owning the file
grpnum	number of the group owning the file
length	length of the file (in bytes)
name	name of the file (not including its path)
owner	name of the user owning the file
ownnum	number of the user owning the file
path	directory where file is located
perm	access permissions of the file
type	type of the file. For VMS, this might be “file” or “directory”
xsum	checksum of the file

User Variables

group	name of the primary group the user belongs to
grpnum	number of the primary group the user belongs to
homedir	the user’s home directory
usernum	number of the user

Group Variables

grpnum	number of the group
user	names of all the users belonging to this group
usernum	numbers of all the users belonging to this group

Global Variables (same for user, group, and file)

item	name of this item (i.e., filename, username, or group name)
generator	same as item, unless the file/user/group name was created via an EVAL command, in which case, the name input to the EVAL subroutine is returned. More about his later.

Compound Information

fstat	File status information. This is a compound quantity and can be reported to the output for future evaluation. However fstat information can not be used in a test.
fstatx	Checksum plus file status information. This is a compound quantity and can be reported to the output for future evaluation. However fstatx information can not be used in a test.
userentry	User status information. This is a compound quantity and can be reported to the output for future evaluation. However userentry information can not be used in a test.
grpentry	Group status infromation. This is a compound quantity and can be reported to the output for future evaluation. However grpentry information can not be used in a test.

EVAL Command

The EVAL command allows a user to perform functionality not directly supported by the CQL language. A predefined set of such routines are supplied, but a user can write his own routines. These routines expect as input a name and a string argument, and can return a comment, warning level, and identifier, in addition to the string value of the EVAL call. Note: the value of an EVAL program is the exit code resulting from its execution.

The EVAL command can be used in several different ways. The EVAL syntax is described below, and examples of each possible EVAL format follow this description.

You can use EVAL to call a subroutine or to execute a program. For the subroutine case, the first argument to EVAL is the subroutine name to execute along with a parenthesized list of zero or more blank delimited arguments. If a second argument is present, then the subroutine is called once with each item in the given list. The EVAL command for programs takes one argument, the execute line.

Suppose function get_paths() returns the list of file names found in a given ascii file. Then, Example 1, below will check the owner of all the files listed in LOGIN.COM and

will report those files which are not owned by root. Files listed in [FRED]LOGIN.COM, etc. are always executed upon start up, and thus are susceptible to trojan attack and thus should have proper owner, permissions, etc.

```
STATUS FILES=
EVAL(get_paths(), {‘[FRED]LOGIN.COM})
     {OWNER!=root}
```

Existence Checks-- Require/Allow/Deny

When running the query tests over a list of objects, you need to first consider whether that object exists. By default, any non-existing objects are ignored. However, their may be cases when you want to ensure that a given file/user/group is present (and be notified if it is not). Similarly you may want to be notified if an object does exist. This can be done by appending the word REQUIRE, ALLOW, or DENY behind the WHICH or STATUS keyword of the query.

The following example checks the owner of file `SYS$SYSTEM:SYSUAF.DAT` if it exists, and reports an error if the password file does not exist.

```
STATUS REQUIRE FILES=‘SYS$SYSTEM:SYSUAF.DAT
{OWNER!=root}
```

The password file needs to be present (example above), but the file `SYS$SYSTEM:SYSUAF.DAT` (example below) does not have to be. Thus, the next example uses the keyword ALLOW (or no keyword).

```
STATUS ALLOW FILES=`SYS$SYSTEM:SYSUAF.DAT { OWNER!=root}
     STATUS FILES=`SYS$SYSTEM:SYSUAF.DAT { OWNER!=root}
```

Except/Probit

Most security policies contain at least a few exceptions. To accommodate this, the CQL language allows you to ignore a specific query item (no testing) or to report the existence of the item as an error

The following example prevents users other than root from having a .rhosts file. Any violation of this policy will be reported.

```
DENY STATUS FILES=~/.rhosts EXCEPT=/.rhosts
```

The next example performs two functions. It first finds the paths defined in the .cshrc file and reports if "." is present. Second, for each non-"." path found, it checks if it has world-writable permissions .

```
STATUS FILES= EVAL(get_paths(),~/.cshrc)
PROHIBIT="."
     { PERM=w_w }
```

Comments, Ids, Warning Levels

Comments, identifiers, and warning levels can be attached to the list of files/users/groups being queried and to the tests being performed. Identifiers can help other tools to distinguish the output from multiple requests. Comments are useful for printing more understandable output reports. Warning levels indicate the severity of the security violation reported (the larger the number, the more severe the problem).

Any passed vulnerability test reports any comment, id, and/or warning level attached to that test. On the other hand, failure of existence tests due to REQUIRE, DENY, PROHIBIT will report any comment, id, and/or warning level attached to the object (file, group, user) list.

```
STATUS FILES= EVAL(get_paths(),'[000000...]LOGIN.COM)
   PROHIBIT="."  WARN=1
   COMMENT=". should not be in search path"

{ PERM=w_w  WARN=2 ID=w_w_path
  COMMENT={"world-writable file accessed
            in " GENERATOR}
}
```

In the above example, if "." is present in the list, then the warning level is 1, but if any file in the list is world writeable, then the warning level is 2.

Note: in the above comment, GENERATOR will expand to the name of the appropriate LOGIN.COM file. For example, if [FRED]LOGIN.COM reads the file /home/me/junk and this junk file is world writeable, then the resulting comment expands to "world-writeable file accessed in `[FRED]LOGIN.COM`".

Reports

The REPORT commands (as shown below) allow you to report individual data items or a list without having to judge it, and are used to gather information which will be analyzed outside of CQL.

```
REPORT DATA = list
REPORT LIST = list
```

The following example reports the file status (inode) and checksum information for the file [FRED]LOGIN.COM.

```
STATUS FILES = { '[FRED]LOGIN.COM }
{ REPORT DATA= FSTATX }
```

The next example makes one report entry containing all the names of "bad users accounts" (as defined by subroutine bad_users).

```
REPORT LIST = EVAL(bad_users())
```

Define

The DEFINE command associates a name to a list so that it can be used later. To use this list, preface the name of the list with @.

The following will report if the files under [FRED1] and [FRED2] are not owned by root.

```
DEFINE fred= {‘[fred1...]*.* ‘[fred2...]*.*}
STATUS FILES=@dotfile { OWNER != root}
```

READ filename

The READ command allows you to read additional files. This command can even be present within an IF/ELSE command. The filename given can have a fully specified path or be relative to current working directory. It can also be relative to the repository of pre-existing CQL scripts.

If/Else

The IF/ELSE command allows you to conditionally execute other commands (such as DEFINE or READ) or query tests. If the IF or ELSE body consists of more than 1 statement, then the body must be bracketed by { }'s. For example:

```
IF (system=unix)
     { READ /usr/local/bin/cqlunix1
     DEFINE sysfile = {/.cshrc /.login}
     }
ELSE
     DEFINE sysfile = { }

STATUS FILES=@sysfile
     {PERM = w_w
     IF (groupopt=on) PERM =g_w
     }
```

The above example checks if the system is a Unix system. If it is, then a special file of Unix checks is read and the list `sysfile` is defined. If the system is not Unix, then a different definition of list `sysfile` is given.

The second IF statement operates on a query test. A check for group writability is performed only if the execute line variable groupopt is "on".

Lists

The Configuration Query Language is a list based language. A list can appear within a query (i.e., the list to run the tests over) and also within the query tests (i.e., testing if a quantity is one of a list of elements --- e.g., GROUP={wheel other}).

A list is one or more of the following components (if 2 or more items are present then the list must be bracketed by { }'s).

name
filename

SCAN filename
^ filename
"quoted string"
@ name (reference an already DEFINEd list)
integer

EVAL command
RECURSE command

A name is an alphanumeric (and can include object variable names like OWNER, in which case the value of the variables is substituted for its name).

A string is quoted with the double quote mark (").

A filename begins with "`".

@ name is a named list defined by the DEFINE command.

The keyword SCAN in front of a filename expands to every file name (but no directory names) recursively found starting from the given file name.

Recurse

The RECURSE command, which is of the form RECURSE(query), allows you to perform a query and to create a list containing the names of those items with security violations. This list can be saved in a DEFINE statement or it can be used immediately.

If subroutine passwd_chk returns the names of user accounts with bad password, then the following example creates a list of bad user accounts.

```
 DEFINE badusers =  RECURSE(STATUS USERS=ALL
{EVAL(passwd_chk())=bad} )
```

A slight variation on the above is to convert the list of bad user accounts into a list of bad group accounts as follows.

```
DEFINE badgrps = RECURSE(GROUP(STATUS
USERS=ALL
     {EVAL(passwd_chk())=bad} ) )
```

You can then use the above list in another query. The following example checks if any user's .login and .cshrc files are writeable by any of the groups in list badgrps***

```
STATUS FILES={~/.login ~/.cshrc}
     { PERM=g_w  &  GROUP=@badgrps}
```

Creating Your Own EVAL Functions

*** Note: you could replace @badgrps in the above example with its definition, i.e. RECURSE(GROUP(...).

In addition to the predefined EVAL functions, you can write your own EVAL functions. To use your own function, you must declare these functions in the cql source code. To do this, you must modify file cqlsrc/usersubs.c by adding two lines to function declare_usersubs().

The first line is the declaration of the function, i.e.

```
char *myfunc();
```

where myfunc is the name of your function.

The second line adds the functions into cql's lookup tables by calling subroutine setfunc() as follows:

```
setfunc("myfunc", myfunc);
```

where the first argument is a string containing the name of the function, and the second argument is the function itself.

Once you finish modifying function declare_usersubs(), you then need to write your EVAL function. This function should also be in file cqlsrc/usersubs.c. The interface to this function will be described shortly. Once all the above is done, then you need to recompile cql. The cql tool will now recognize and be able to execute your function upon request.

The arguments to an EVAL function are as follows:

```
evalfunc (name1, args1, args2, name2, comment, eid, warn,
          errstr)
```

The input arguments are

name1:	(char *) name of object to look at
args1:	(char *) string containing arguments (supplied in the EVAL call in the script)
args2:	(char *) string containing default arguments (supplied at the cql execute line using the -t option)
name2:	(char *) name of object to look at

Output arguments are

comment:	(char **) array of strings describing security problems found. If *comment is not NULL, then the final string in the array must end in an **extra** '\0'. For example: *comment="Bad permission\0\0".
eid:	(char **) identifying string used when grouping the found security problems into a final report. If no eid value is returned for a problem finding EVAL function, then any problems found

would be grouped under the name of the EVAL function and its args1 and args2 values.

warn: (int *) integer value denoting severity of the problem found

errstr: (char **) *errstr should be NULL, unless bad arguments to the function are used. In which case, set *errstr to a warning message, and a cql language syntax error will eventually be issued.

Returned value is

(char *) the value of the EVAL function is a list of 0 or more blank delimited items -- such as names, numbers, etc. For example, most of the predefined EVAL functions either return a list of file names or the string "good" or "bad".

There are two name input arguments to the EVAL function. Usually only one of these two arguments is used and the other is a dummy. These names arguments correspond to the two main types of EVAL commands, as illustrated below.

```
STATUS FILES=EVAL(myfunc(),
                  {‘[FRED]LOGIN.COM ‘[FRED]SPECIAL.COM}
                  { some tests }
```

and

```
STATUS FILES={`SYS$SYSTEM:SYSUAF.DAT
                `SYS$SYSTEM:PARAMS.DAT}
      {EVAL(newfunc())=bad}
```

The first example above, illustrates what I have previously referred to as a list function. The function myfunc() will be called twice, first with argument name2 set to `[FRED]LOGIN.COM and then with name2 set is `[FRED]SPECIAL.COM. Argument name1 is a dummy argument in this case.

The second example above illustrates a check function (i.e. checking if some security violation is present). The function newfunc() will be called twice. Once with argument name1 set to SYS$SYSTEM:SYSUAF.DAT and once with name1 set to SYS$SYSTEM:PARAMS.DAT.

Coding examples follow for list EVAL functions and check EVAL functions.

List EVAL Function Coding Example

The value returned by a predefine list function is a list of other objects (files, users, groups). Since no evaluation is done, these routines generate no comments, warning levels, or identifiers.

```
char *listfunc (dummy, args1, args2, name, comment,
                eid, warn, errstr)
```

```
{
char *result, *newname;
char line[1024];
line[0]='\0';

/* parse args1 and args2 for all values they */
/* may contain -- argument values will be blank */
/* delimited i.e. args1="-s 1 -x" */]

/* if (args1 contains an unknown option) goto bad */

/* NOTE:  comment, eid, and warn are not set/used */

/*NOTE: name may refer to a non-existent object*/
fid = fopen(name, "r")
if (fid==NULL) return(NULL);

while (fgets(line, 1024, fid) != NULL)
  {  newname = process_line(line);
     strcat(line," ");
     strcat(line, newname);
  }
result=malloc(strlen(line)+1);
strcpy(result, line);
return(result);   /*return dynamic not static memory*/

bad:
  *errstr=malloc(13);
  strcpy(*errstr, "Bad options");
  return(NULL);
}
```

Predefined Check Functions

The value returned by a predefine check function is an evaluation of the object (file, user, group) being checked. Usually this evaluation is either "bad" or "good", but sometimes it may be a list of the bad items found. If a bad state is found, a predefined check function generates one or more comments and often generates identifiers.

As previously mentioned, comments are returned as an array of strings. To help manage this, three comment routines are provided. They are

```
char * com_add(string, delimiter, newstring)
```

where newstring is appended to the end of string, and the character delimiter is then added to the end of this.

To translate concatenated strings into an array of strings, call

```
void com_arry(string, delimiter)
```

This routine will turn all characters matching character delimiter into '\0'. Call this string once when you are all done concatting your comments together. Make sure argument

delimiter is not a character which naturally occurs in your comments. The delimiter character in com_arry should match the delimiter character in com_add.

There may be times when you want to use the incoming value of *comment and concat to this. However, this incoming value would be an array of strings, but com_add() expects just a single string. To translate from an array of strings into a string, call

```
void com_unarry(string, delimiter)
```

where the arguments to com_unarry are the same as the arguments to com_arry.

Given the above subroutine definitions, a coding example of a check EVAL function is as follows:

```
char *checkfunc (name, args1, args2, dummy, comment,
                 eid, warn, errstr)
{
char *result, *newname;
char line[1024];
line[0]='\0';

/* parse args1 and args2 for all values they */
/* may contain -- argument values will be blank */
/* delimited i.e. args1="-s 1 -x" */]

/* if (args1 contains an unknown option) goto bad */

result=malloc(15);
strcpy(result, "good");

*eid = malloc(25);
strcpy(*eid, "configuration test");

com_unarry(*comment, '$');

/*NOTE: cql ensures that name refers to an existing */
/*      object.  If not, then the EVAL function is not   */
/*      called.  This is different than for list EVAL calls
*/
fid = fopen(name, "r")
if (fid==NULL) go to bad;

while (fgets(line, 1024, fid) != NULL)
  {  if (is_bad_line1(line))
        { *comment=
                com_add(*comment, '$', "bad type 1");
          strcpy(result, "bad");
         }
     else if (is_bad_line2(line))
        { *comment=
                com_add(*comment, '$', "bad type 2");
          strcpy(result, "bad");
         }
  }
```

```
com_arry(*comment, '$');
return(result);

bad:
  *errstr=malloc(13);
  strcpy(*errstr, "Bad options");
  com_arry(*comment, '$');
  return(NULL);
}
```

Report Generator (RG)

The SPI Report Generator is used to convert "CORF" (Common Output Report Format) files into variably-formatted final reports. A corresponding parameter file must be supplied to direct the re-formatting. A single-character field delimiter for the given datafile must be supplied. Data is in CORF form if it is ASCII text, newline-delimited records comprised of "delimiter"-delimited fields, where delimiter is a single ASCII character.

Command Line Options

```
rg  [-h] datafile  parameterfile  [-d<delimiter>]
```

-h	display command line usage (SYNOPSIS)
-d<delimiter>	delimiter is a single ASCII character representing the primary field delimiter for the supplied datafile. If no delimiter is supplied, colon ':' is assumed.
datafile	This file contains the data to be formatted. The data must consist of newline-delimited records with fields delimited by a single delimiter character. The records may be considered to have a common part and a variant part. The common part is comprised of those fields representing meaningful attributes across all records in the datafile, and are thus candidates upon which to sort/section the report, if desired. (see SORT specification in the parameterfile discription.)
parameterfile	This file directs the datafile reformatting.

Parameter Files

Lines beginning with '#' or whitespace are treated as comments (ignored.) All other lines must begin with one of

TITLE SUBTITLE FIELD SORT SUBSORT OMIT PUT_ORDER LABEL RESECTION

Of these, FIELD SORT SUBSORT OMIT PUT_ORDER and LABEL specify formatting which will be applied to ALL entries in the datafile, with the exception of entries that are temporarily put aside by the OMIT specifications. The specified formatting remains in force until a line beginning with the keyword RESECTION is encountered, at which point any OMIT-ed datafile entries are rejoined with all other datafile entries, and a new round of formatting may be specified.

With each round of formatting, the recursive SORT and SUBSORT specifications will be applied to all (non OMIT-ed) datafile entries. The SORT specifications allow the remaining datafile entries to be recursively sorted and sectioned on one or more given sortfields, dividing the entries into nested subsets. The innermost subsets will contain identical values for the selected sortfields. A degree of variable formatting within each round is then

provided by the PUT_ORDER and LABEL specifications. Each of these allow conditioning on the values of the sortfields, and provide variance in the fields to be output, the field order, and the labeling of column headers, as appropriate for each of the innermost data subsets. The SUBSORT specification allows an additional recursive sort to take place across all subsets, but the entries are simply sorted (not sectioned,) and the field(s) upon which the SUBSORT takes place are not considered as sortfields for the purposes of conditioning.

Below are details for each of the format specifications.

TITLE SECTION: This is typically a single line indicating a desired report title. The title specified will be repeated automatically at each RESECTION. The format is

```
TITLE:<Your Title Goes Here>\\n\\n
```

Example,

```
TITLE:********* The V.I.P. Report **********\\n\\n
```

SUBTITLE SECTION: This is typically a single line indicating a desired report subtitle (typically varying across RESECTIONS). The format is

```
SUBTITLE:<Your Subtitle Goes Here>\\n\\n
```

FIELD SECTION: This section defines FIELDNAMES for selected fields in the datafile. FIELD lines have the format

```
FIELD:<fieldnumber>:<fieldname>[:<printformat>]
```

The given <fieldname> is used as a token to identify selected datafile fields for operations described in the remaining sections. The <printformat> specification is only used if the given field is among those selected to section the output report (see SORT SECTION.) Not all of the datafile's fields need to have defined fieldnames.

For example, some fields in SYS$SYSTEM:SYSUAF.DAT might be specified by

```
FIELD:1:AccountName:--- The %s Account ---\\n\\n
FIELD:3:UserID
FIELD:4:GroupID
```

SORT SECTION: This section defines the named field(s) whose values will be used to "section" the formatted report. Each line has the form

```
SORT:<sort_level>:<fieldname>[:<grouping...>]
```

NOTE: A SORT assignment should only be made to a subset of those fields which represent meaningful attributes across ALL data record instances.

The <sort_level> must be one of { 1,2,3,...9 }, begin with 1, and increase by 1 for each additional SORT specification. To continue with the SYS$SYSTEM:SYSUAF.DAT example, if you wanted to group the entries according to the user's group ID, you would specify

```
SORT:1:GroupID
```

Each section will be identified by the section title. If you were careful to supply a <printformat> for GroupID in the FIELD definition, the current GroupID value would be used to substitute for the '%s' in the <printformat>, and the title would look like

```
"*** GroupID = xxx ***"
```

otherwise it will simply state

```
"Section = xxx:"
```

If you want to further divide each of these sections into subsections, according to the value of (say) the user's GroupID, you would use

SORT:1:GroupID

SORT:2:UserID

SUBSORT SECTION: This section defines the named field(s) whose values will be use to further recursively sort the output lines. Each specification has the form

```
SUBSORT:<sort_level>:<fieldname>[:<grouping...>]
```

NOTE: A SORT assignment should only be made to a subset of those fields which represent meaningful attributes across ALL data record instances.

The <sort_level> must be one of { 1,2,3,...9 }, begin with 1, and increase by 1 for each additional SUBSORT specification. See SORT SECTION (above) for a description of the optional 'grouping'.

OMIT SECTION: All records with a selected value in a single conditioning field can be temporarily set aside. The OMIT specification is

```
OMIT:<fieldname><relation><value>
```

where 'relation' may be any one of { =, <, >, <=, >=, != }. The records set aside are not available for formatting or output until a RESECTION occurs.

PUT_ORDER SECTION: For each section[subsection,...] of the report, those records of the datafile which meet the criteria for being in that part of the report are now to be printed, each one to a line. Here, for each such section, you may specify which record fields are to be output, and in what order. The format of the PUT_ORDER specification is

```
PUT_ORDER:<fieldnum(s),...]>:<condition(s),...]>
```

where fieldnums are the field numbers of fields to be reported, and conditions have the form <fieldname><relation><value>.

NOTE: the fieldnames referenced in the above conditions are restricted to those fields for which sort/sectioning has been specified.

LABEL SECTION: In a fashion similar to the PUT_ORDER section described above, one can specify individualized column labels based upon the current section of the report. The format is

```
LABEL:<fieldnum>:<label>:<format>:<condition(s),...>
```

NOTE: The <format> (field width, left/right justify, etc) is applied not only to the column label, but also to all data in that column.

As an example, assume a datafile contained facts about files, users, and groups. Assume that field number 5 (say it has been given the defined fieldname TYPE) indicated this division by the field values FILE, USER, and GROUP. Assume also that field number 7 holds the name of the listed files (or users, etc.) If you have not specified a sort/sectioning upon the TYPE field, then files, users and groups will be mixed together, and the best label you could supply for field 7 might be "Subject_Name". I.e.,

```
LABEL:7:Subject_Name:%-20s :
```

If, however, you had specified (at some level) a sort/sectioning upon field 5 (the TYPE field), then the entries for files will be listed together in one section, entries for users listed in another, etc. It would then make more sense to specify the label(s) for field 7 by

```
LABEL:7:File_Name:%-20s :TYPE=FILE
LABEL:7:User_Name:%-20s :TYPE=USER
LABEL:7:Group_Name:%-20s :TYPE=GROUP
LABEL:7:Subject_Name:%-20s :
```

Note that you may still leave an entry for "Subject_Name" with no given conditions. This specification will be picked up in the event that either TYPE is not one of FILE, USER, or GROUP, or in the event that you failed to specify a sort/sectioning upon the TYPE field. Note also the format "%-20s ". This sets aside 20 characters for the item name, left justified. The blank space before the next ':' forces at least one space after the 20, so in the event that the data runs to more that 19 characters, a blank space still separates that entry from subsequent data fields.

Technical Information Department • Lawrence Livermore National Laboratory
University of California • Livermore, California 94551